🚀 JavaScript Strings & Date Object

(Detailed notes yet beginner friendly)

String - Introduction

What is a String and why does it exist?

In programming we constantly deal with text data such as:

- Names
- Messages
- Emails
- Passwords
- URLs
- JSON data

Computers fundamentally understand numbers, not text.

So programming languages created a special data type called STRING which stores sequences of characters.

Examples of strings:

"Hello"
"JavaScript"
"Rajesh"
"Hello World"

A string is simply a sequence of characters stored together.

String - Key Characteristics

Key Characteristics of Strings

1. Strings are Primitive Values

In JavaScript, strings are primitive data types.

```
let name = "Alex"
console.log(typeof name)  // Output "string"
```

Even though they behave like objects sometimes, internally they are still primitive values.

2. Strings are Indexed from Zero

Each character inside a string has a position.

Example

```
let word = "JavaScript"

J a v a S c r i p t
0 1 2 3 4 5 6 7 8 9
```

We can access characters using indexes.

```
let word = "JavaScript"

console.log(word[0])        //Output → J
console.log(word[4])        //Output → S
```

String - Key Characteristics

2. Strings are "Object-like" Primitives

Even though strings are primitive values, JavaScript temporarily converts them into String objects when methods are used.

This allows strings to use methods like:

.toUpperCase()
.includes()
.slice()

Example

```
let text = "hello"

console.log(text.toUpperCase());  //Output → HELLO
```

Creating Strings

JavaScript provides three main ways to create strings:

1. **Single Quotes**
2. **Double Quotes**
3. **Template Literals**

Single Quotes and Double Quotes

Single quotes allow double quotes inside.

```
let text = ' Hello "World"  '
```

Double quotes allow single quotes inside.

```
let text = "  It's JavaScript  "
```

Both behave exactly the same in JavaScript.

Template Literals (Modern Way)

Template literals use backticks (`)

They allow embedding variables directly inside strings.

Example

```
let name = "Alex"
let message = `Hello ${name}`
console.log(message)
```

Output

```
Hello Alex
```

Template literals are commonly used in modern JS.

Core Properties and Concepts

A. The .length

Every string has a. length property that tells you how many characters it contains.

```
let greeting = "Hello";
console.log(greeting.length);         // Outputs: 5

let emptyString = "";
console.log(emptyString.length);    // Outputs: 0
```

B. The GOLDEN Rule: Strings are Immutable

This is the most critical concept. You cannot change a string in place. Any method that appears to modify a string will always return a brand new string, leaving the original untouched.

```
let name = "alex";

// Let's try to change the first character.

name[0] = "A";                  // This will FAIL silently. It does nothing.
console.log(name);              // Outputs: "alex" (The original string is
unchanged)

// Let's use a method that "changes" the string.

let upperName = name.toUpperCase();
console.log(upperName);   // Outputs: "ALEX" (This is a NEW string)
console.log(name);              // Outputs: "alex" (The original is still
unchanged)
```

Common and Essential String Methods

These methods are your primary tools for working with strings.
Remember, they all return new strings.

A. Changing Case

- **.toUpperCase()** : Returns a new string with all characters in uppercase.
- **.toLowerCase()** : Returns a new string with all characters in lowercase.

```
const earth = "I love Mother Earth";
const UpperCase = earth.toUpperCase();  // "I LOVE MOTHER EARTH"
const LowerCase = earth.toLowerCase();  // "i love mother earth"
```

B. Finding Substrings

- **.indexOf(substring)** : Returns the index of the first occurrence of a substring. If the substring is not found, it returns 1.
- **.lastIndexOf(substring)** : Returns the index of the last occurrence of a substring. Returns 1 if not found.
- **.includes(substring) (ES6)** : Returns a boolean (true or false) indicating if the string contains the substring. This is often more readable than indexOf.

Common and Essential String Methods

```
let sentence = "The quick brown fox jumps over the lazy fox.";

console.log(sentence.indexOf("fox"));          // 16 (the first one)
console.log(sentence.lastIndexOf("fox"));      // 40 (the last one)
console.log(sentence.indexOf("cat"));          // -1 (not found)
console.log(sentence.includes("jumps"));       // true
console.log(sentence.includes("cat"));         // false
```

C. Extracting Substrings

- **.slice(startIndex, endIndex)** : Extracts a section of a string and returns it as a new string.

1. **startIndex** : The index where the extraction begins (inclusive).
2. **endIndex** : The index where the extraction ends (exclusive - it does not include this character).
3. You can use negative indices, which count from the end of the string.

```
let text = "JavaScript";

console.log(text.slice(0, 4));        // "Java" (from index 0 up to, but not including, 4)
console.log(text.slice(4));           // "Script" (from index 4 to the end)
console.log(text.slice(-6));          // "Script" (the last 6 characters)
```

- **.substring(startIndex, endIndex)** : Similar to .slice(), but it doesn't accept negative indices.
- **.substr(startIndex, length)** : (Deprecated - Avoid). It works with a start index and a length, which is different from the others. Use .slice() instead.

Common and Essential String Methods

D. Replacing Substrings

- **.replace(searchValue, newValue)** : Finds a searchValue and replaces it with newValue.
- NOTE: .replace only replaces one value in the entire string
- **.replaceAll(searchValue, newValue) (ES2021)** : Replaces all occurrences. This is the modern, easy way to do a global replace.

```
let greeting = "hello world, hello there";

// Replaces only the first "hello"
let newGreeting = greeting.replace("hello", "hi");
console.log(newGreeting);          // "hi world, hello there"

// Replaces all "hello"s
let allNewGreeting = greeting.replaceAll("hello", "hi");
console.log(allNewGreeting);      // "hi world, hi there"
```

E. Removing Whitespace

- **.trim()** : Removes whitespace from both the beginning and end of a string.

- **.trimStart()** / **.trimEnd()** : Removes whitespace from only the start or end.

```
let userInput = " my-username@example.com ";
let cleanedInput = userInput.trim(); // "my-username@example.com"
```

Common and Essential String Methods

F. Splitting a String into an Array

- **.split** (separator) : Splits a string into an array of substrings, using the separator to decide where to split. This is incredibly useful.

```
let csvData = "item1,item2,item3";
let items = csvData.split(",");            // ["item1", "item2", "item3"]

let words = "The quick brown fox";
let word_array = words.split(" ");     // ["The", "quick", "brown", "fox"]

let letters = "abc";
let letter_array = letters.split("");     // ["a", "b", "c"]
```

Date - Introduction

Why do we need Date ?

Applications constantly deal with time:

- Login timestamps
- Post creation time
- Scheduling tasks
- Timers
- Analytics

So JavaScript provides a built-in Date object to handle time.

The Fundamental Truth of Date

Internally Date is just a BIG NUMBER. That number represents milliseconds passed since a starting reference point called **Unix Epoch**

The **Unix epoch** is the foundational time, **00:00:00 UTC on January 1, 1970**, used by computing systems to calculate time by counting seconds elapsed since that moment (ignoring leap seconds).

- Every date you create is just a wrapper around this timestamp.
- This makes it easy to compare dates and calculate durations.

```
console.log(Date.now())    // 1710500000000 (example)

// That number represents milliseconds
since January 1, 1970.
```

Year 2038 Problem

Early Unix systems stored time as a signed 32-bit integer.

This value represents the number of seconds since the Unix Epoch (Jan 1, 1970).

Maximum value:
2,147,483,647 seconds

This corresponds to: **January 19, 2038**
After this point, the value overflows and wraps around to:
-2,147,483,648 which represents: **December 13, 1901**

This happens due to integer overflow.

Important:

This problem affects systems using **32-bit** time in seconds (like C/Unix).

JavaScript is NOT affected because it uses **64-bit** floating-point numbers and stores time in milliseconds.

Solution:

Modern systems use 64-bit integers, allowing time to be stored for billions of years into the future.

UTC and it's Importance

What is UTC (stands for Coordinated Universal Time)

Coordinated Universal Time (UTC) is the primary time standard globally used to regulate clocks and time. It establishes a reference for the current time, forming the basis for civil time and time zones. UTC facilitates international communication, navigation, scientific research, and commerce.

Why do servers use UTC time?

In real-world applications, users exist across different time zones:

India (IST) → UTC +5:30
USA (EST) → UTC -5
UK (GMT) → UTC +0

If every system used local time, it would create massive confusion.

Example problem:

User in India creates a post at 10:00 AM IST
Server in USA stores time as 10:00 AM (EST)

Now the same event has TWO different meanings.

This leads to:

- Incorrect timestamps
- Wrong ordering of events
- Bugs in scheduling systems
- Data inconsistency

UTC and it's Importance

The Solution → Use UTC

UTC (Coordinated Universal Time) is a global standard time.
It does NOT change based on:

- **Location**
- **Time zones**
- **Daylight Saving Time (DST)**

All servers store time in UTC to maintain consistency.

How it works in practice ?

1. Server stores time in UTC
2. Client converts it to local time

Example:

```
let now = new Date();
console.log(now.toISOString()); //2026-03-17T12:30:00.000Z  (UTC time)
console.log(now.toLocaleString());  //Shows time in user's local timezone
```

Why UTC is IMPORTANT ?

1. Consistency → Same time reference across all systems
2. No DST Issues → UTC never changes like local time zones
3. Easy Comparison → Timestamps can be compared directly
4. Global Applications → Works correctly for users worldwide

Golden Rule

Always store time in UTC
Convert to local time ONLY for display

Creating Date Object

You can create a Date object using **new Date()** constructor

A. Create a Date for the Current Moment

If you provide no arguments, you get the current date and time from the user's device.

```
// Creates a Date object for right now.

const now = new Date();
console.log(now);          // e.g., Tue Mar 17 2026 10:50:00 GMT+0530 (India Standard Time)
```

B. Create a Date from a Timestamp

You can create a date from that big "milliseconds" number we talked about. This is how servers and databases often communicate time.

```
// Date.now() is a quick way to get the CURRENT timestamp as a number.

const currentTimestamp = Date.now();   // e.g., 1773724008326

console.log(currentTimestamp);       // Now, let's create a Date object from that number. const da = new Date(1773729008207);

console.log(da);                     // This will show the date and time corresponding to that timestamp.
```

Creating Date Object

C. Create a Specific Date (The Recommended Way)

This is the most reliable way to create a specific date. You provide the components as numbers in this order:
year, month, day, hours, minutes, seconds, ms

```
// new Date(year, month, day, hours, minutes, seconds, ms)

const myDate = new Date(2026, 2, 17, 11, 15, 57, 658);

console.log(myDate.toString());     // Tue Mar 17 2026 11:15:57 GMT+0530 (India Standard Time)
```

🚨 CRITICAL BUG: Months are Zero-Indexed! 🚨

This is the most common source of bugs with JavaScript dates.

0 = January
1 = February
2 = March (done in above example)
...
8 = September
9 = October
10 = November
11 = December
NOTE: Days are start normal from 1 index but year does NOT!

WHY this problem exists?
It is primarily due to historical reasons and a design decision to align with zero-based array indexing

Date Object (Getters)

Getting Information from a Date Object (Getters)

Once you have a Date object, you can extract its individual parts. These methods return values based on the user's local timezone.

```
const now = new Date();     // Let's assume it's Mar 17, 2026, Tuesday

console.log(now.getFullYear()); // 2026 (The full 4-digit year)
console.log(now.getMonth());    // 2 (Remember, 0-indexed, so 2 is March)

console.log(now.getDate());     // 17 (The day of the month, 1-31)
console.log(now.getDay());      // 2 (The day of the week: 0=Sunday, 1=Monday, ..., 4=Thursday)

console.log(now.getHours());    // e.g., 11 (The hour, 0-23)
console.log(now.getMinutes());  // e.g., 15 (The minute, 0-59)

console.log (now.getMilliseconds);  //e.g., 548 (Milliseconds, 0-999)

console.log(now.getTimezoneOffset()); //Timezone returns a number i.e.,Returns the difference (in minutes) between UTC and local time.

Positive → Local time is behind UTC
Negative → Local time is ahead of UTC

 E.g., -330, (it is Indian time offfset means it is ahead 330 minutes of UTC and it we subtract 330 minutes then we'll get UTC, that's why it is in minus value)
```

Date Properties

Changing a Date: Setters and Mutability

Date objects are mutable, meaning you can change their value in place using "setter" methods.

```
const date = new Date(1773729008207);    // Creates a specific date

console.log("Before:", date); // Let's change the month to May (index 4)

date.setMonth(4);

console.log("After:", date);    // The ORIGINAL 'date' object has been changed.
```

Date Auto-Correction (Rollover)

The Date object is smart about invalid dates. If you give it a value that's out of range, it will "roll over" to the next logical date.

Your example new Date(2026, 1, 30) is a perfect demonstration. You are asking for February 30th, 2026.

```
// 2026 is not a leap year, so February has 28 days.
// You are asking for Feb 28 + 2 days.

const a = new Date(2026, 1, 30);      // The date object auto-corrects to March 2nd, 2026.
console.log(a);                       // Outputs: Sun Mar 02 2026 ...
```

Formatting Dates for Display

A raw Date object isn't very user-friendly.
You must format it into a string for display.

```
const now = new Date();

 // .toString() is the default, verbose format in the local timezone.
console.log(now.toString());
// e.g., Mon Oct 28 2024 04:15:00 GMT+0530 (India Standard Time)

// .toDateString() gives you just the date part.
console.log(now.toDateString());
// e.g., Mon Oct 28 2024

// .toISOString() gives the universal, unambiguous UTC time. The 'Z'
means UTC. // This is what you send to servers.
console.log(now.toISOString());
 // e.g., 2024-10-27T22:45:00.000Z

// .toLocaleString() formats the date and time in a way that is natural
// for the user's region and language. BEST for user display.
console.log(now.toLocaleString());
// e.g., in India: 28/10/2024, 4:15:00 am // e.g., in the US:
10/27/2024, 4:15:00 PM
```

Major Problems of Date Object

The Date API has many issues:

1. Months start at 0, but days start at 1
2. Mutable (can be changed accidentally)
3. Weird quirks and bugs
4. Difficult formatting

The Solution:

Because the Date API has several limitations, developers use libraries:

Day.js (lightweight, 2KB)
date-fns (functional, tree-shakeable)
Luxon (modern, powerful)

Important:
JavaScript Date uses the system's local timezone by default, which can lead to unexpected behavior when working with global apps.

Temporal API

JavaScript is getting a new date/time API called Temporal that fixes all these issues:

Key Features

Immutability: All operations on Temporal objects return a new instance, preventing unintended side effects and making code more predictable.

Explicit Time Zones: It offers first-class support for time zones and handles complexities like Daylight Saving Time (DST) safely, which the Date object struggled with.

Comprehensive Types: Instead of a single, ambiguous Date object, Temporal provides distinct classes for different use cases, making code clearer.

Precision: Temporal supports nanosecond precision, which is useful for applications requiring high accuracy.

1-Based Months: Unlike the Date object where months are 0-based, Temporal uses 1-based indexing (January is month 1), which is more intuitive.

Internationalization: It has built-in support for non-Gregorian calendars.

Current Status: Temporal is currently in Stage 3 and should be available in browsers/Node.js soon!